A 64-bit Stream Processor Architecture for Scientific Applications

Xuejun Yang, Xiaobo Yan, Zuocheng Xing, Yu Deng, Jiang Jiang, and Ying Zhang
National Laboratory for Paralleling and Distributed Processing, School of Computer,
National University of Defense Technology, Changsha, Hunan, 410073, P.R. of China
{xjyang, xbyan, zcxing, yudeng, jiangjiang, zhangying}@nudt.edu.cn

ABSTRACT
Stream architecture is a novel microprocessor architecture with wide application potential. But as for whether it can be used efficiently in scientific computing, many issues await further study. This paper first gives the design and implementation of a 64-bit stream processor, FT64 (Fei Teng 64), for scientific computing. The carrying out of 64-bit extension design and scientific computing oriented optimization are described in such aspects as instruction set architecture, stream controller, micro controller, ALU cluster, memory hierarchy and interconnection interface here. Second, two kinds of communications as message passing and stream communications are put forward. An interconnection based on the communications is designed for FT64-based high performance computers. Third, a novel stream programming language, SF95 (Stream FORTRAN95), and its compiler, SF95Compiler (Stream FORTRAN95 Compiler), are developed to facilitate the development of scientific applications. Finally, nine typical scientific application kernels are tested and the results show the efficiency of stream architecture for scientific computing.

Categories and Subject Descriptors
C.1.2 [**Computer Systems Organization**]: Processor Architectures – *Single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms
Design

Keywords
stream processor, architecture, program language, compiler, scientific application, high performance computing

1. INTRODUCTION
Stream processors [11, 13, 7, 20, 3, 22, 6] (e.g. Imagine [11, 13] from Stanford University) have demonstrated significant performance advantages in the domains such as signal processing [9], multimedia and graphics [18]. Yet, it has not been sufficiently validated whether stream processor is efficient for scientific computing. To address this issue, we carry out this study on architecture, programming language and compiler.

ISCA'07, June 9-13, 2007, San Diego, California, USA.

We design and implement a 64-bit stream processor, FT64 (Fei Teng 64), for scientific computing. FT64's instruction set architecture is optimized for scientific computing. Kernel-level instructions are in the form of VLIW, and nearly half of them are related with 64-bit double-precision floating-point multiply-accumulate operations. FT64's processing unit consists of four ALU clusters, and each of them contains four floating-point multiply-accumulate (FMAC) units. FT64's peak performance reaches 16GFLOPS.

We put forward two kinds of communications: message passing and stream communications. Stream communication is shared memory communication based on streams which is similar to Imagine's. Based on the two kinds of communications, we implement an interconnection for designing FT64-based high performance computers.

We design a stream programming language, Stream FORTRAN95 (SF95), for FT64. SF95 extends FORTRAN 95 with ten compiler directives to facilitate the development of scientific stream programs. We also develop a compiler for SF95, Stream FORTRAN95 Compiler (SF95Compiler), which adopts some stream architecture oriented optimizing techniques, including loop streamizing, vector streamizing, stream reusing, parameter reusing, *if* conversion, reduction recognition and kernel fusion, etc.

We perform experiments on FT64 with nine typical scientific application kernels, including three NPB benchmarks (EP, MG and CG), a SPEC2000 benchmark (Swim), and five important scientific application kernels (FFT, Laplace, Jacobi, GEMM and NLAG-5). The results show that for these scientific application kernels except CG, FT64 performs equal or better to Itanium 2.

The remainder of this paper is organized as follows: Section 2 presents related work; Section 3 overviews FT64 stream processing system; the implementation details of FT64 processor and SF95Compiler are given respectively in Section 4 and Section 5. The experiments are shown in Section 6. In Section 7, we conclude the paper and discuss some future work.

2. RELATED WORK
Currently, research on stream processor mainly focuses on processor architecture, programming model and compiler design, etc. But, most efforts are made for media applications, with insufficient supports for scientific applications.

2.1 Stream Processor
Stream models first appeared in Hoare's communicating sequential processor (CSP) [10], followed by David May's efforts in OCCAM [17]. Recently, with the improvements in IC technologies, stream models are further studied and applied in the domains of graphics, multimedia and signal processing, where many architectures and processors supporting stream models have emerged, such as Cheops [3], SCORE [6], Imagine, RAW [22], VIRAM [14], TRIPS [5] and

LEAP [8]. In some work, stream models have been applied to scientific computing, such as Merrimac [7] and MASA [25]. In addition, Cell [19] also supports stream models and is claimed to have tremendous potential for scientific computing. For scientific applications, published stream processors have the following disadvantages: First, the 64-bit double-precision calculation capability is not powerful enough. Stream processors for graphics, multimedia and signal processing are mostly 32-bit. Cell's 64-bit double-precision calculation capability also needs improvements [26]. Second, their supports for FMA (fused multiply-add) operations are not sufficient for scientific computing. Currently, Merrimac and Cell contain multiple multiply-add units, whereas Merrimac's implementation has not been announced by far and Cell has only eight such units. Finally, the capability of inter-chip communication should be further enhanced. Only stream communication is implemented in Imagine, which limits the scalability of multi-chip systems to some degree. These disadvantages show that stream architecture needs further improvements for scientific computing.

2.2 Stream Language and Compiler

The representative stream programming languages include StreaMIT [23], StreamC/KernelC [15] and Brook/BrookTran [4]. StreaMIT is a stream programming language developed by MIT for RAW, which is extended from a subset of Java syntax. Its compiler, StreamIt [24], enables a set of program transformations based on the state space representation [1], including combination of adjacent filters, elimination of redundant states and so on. StreamC/KernelC is a two-level programming language designed by Stanford University for Imagine. The optimization techniques of its profile guided compiler mainly focus on stream scheduling [12], memory access scheduling [21] and communication scheduling [16], etc. Brook/BrookTran is another stream programming language designed by Stanford University for Merrimac. It extends the C/FORTRAN programming language with a few new modifier keywords such as *stream* and *kernel* for specifying streaming primitives. In addition, Brook introduces a runtime library which allows for creation and manipulation of streams.

However, for scientific computing, these languages and compilers have some drawbacks. There is not sufficient support for FORTRAN, which is quite popular in this domain. And they are generally trying to establish a new programming style, which is quite different from the common style known by programmers and incurs heavy burden on inheriting legacy codes.

3. SYSTEM OVERVIEW

FT64 supports stream programming model [15]. In this model, an application is represented by a set of computation kernels which consume and produce data streams. Each data stream is a sequence of data records of the same type, which can be classified into two kinds: basic streams and derived streams. Basic stream defines a new sequence of data records while derived stream refers to all or part of an existing basic stream. Each kernel is a program which performs the same set of operation on each input stream element and produces one or more output streams. Stream applications consist of stream-level programs and kernel-level programs. A stream-level program specifies the order in which kernels execute and organize data into sequential streams that are passed from one kernel to the next. A kernel-level program is structured as a loop that processes element(s) from each input stream and generates output(s) for each output stream.

A FT64 stream processing system consists of FT64 architecture, system interconnection, programming language and its compiling tools.

3.1 FT64 Architecture

FT64 is mainly composed of a stream controller (SC), a stream register file (SRF), a micro controller (UC), four ALU clusters, a stream memory controller (SMC), a DDR memory controller (DDRMC), a host interface (HI) and a network interface (NI), as illustrated in Figure 1.

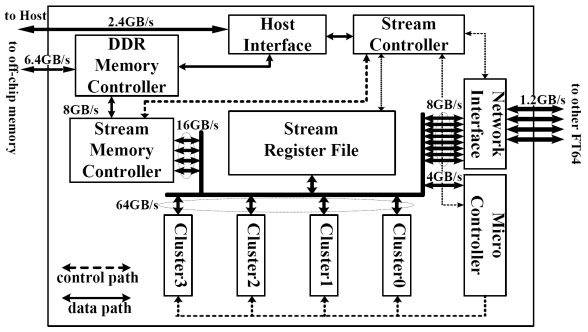

Figure 1. Block diagram of FT64.

FT64 runs as a coprocessor to a host through the HI. The host executes stream-level programs, loads the streams and kernel-level programs to FT64's off-chip memory, and sends stream-level instructions to the SC. The SC executes stream-level instructions. First, it loads a kernel-level program from the off-chip memory to UC's instruction memory through the SRF. Then, the SC loads streams from the off-chip memory to the SRF. When the kernel-level program and streams are ready, the SC will start the execution of the kernel-level program in the UC's instruction memory, and the UC controls the 4 clusters to process data in a SIMD fashion. When the computation is finished, the SC will transfer the output stream stored in SRF to the off-chip memory or to other FT64's SRF through the NI directly.

3.2 System Interconnection

We bring forward two kinds of communications: stream and message passing communications, based on which an interconnection is designed for FT64-based high performance computer system as depicted in Figure 2. This system includes multiple modules, and each of them contains 1 host and 8 FT64s.

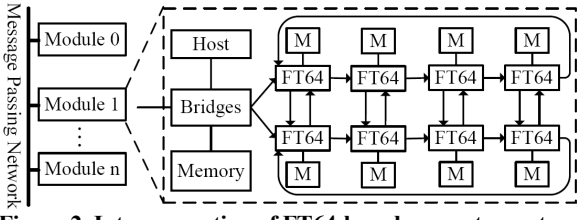

Figure 2. Interconnection of FT64-based computer system.

Within one module, a 2-dimensional mesh network connects multiple FT64s through their NIs. The mesh uses stream communication, which is distributed shared memory communication based on streams, to transfer streams between the SRFs of two FT64s.

Within one module, the host communicates to the FT64s through the HIs by message passing. The HI supports RRA

(Remote Register Access) and DBT (Data Block Transfer) protocols. RRA protocol is used for the host to read/write the registers in FT64s, and DBT protocol can be used to transfer the block data between the host and FT64s in DMA-like mode.

Multiple modules are connected to a message passing network, in which the DBT protocol is used for the FT64s or hosts in different modules to communicate to each other.

3.3 Stream Programming Language

The traditional stream languages are not suitable for scientific computing. To map scientific application programs to FT64 more effectively, we design a new stream programming language, SF95. The details of its compiler will be discussed later in Section 5.

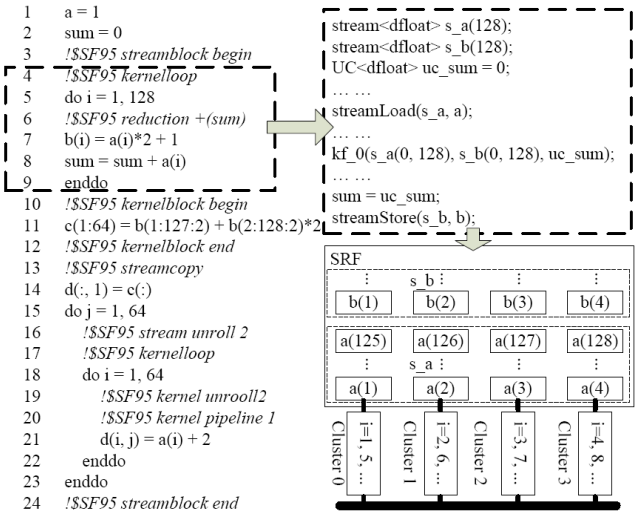

Figure 3. A SF95 program and its execution on FT64.

SF95 adopts a compiler directive based programming model. It extends FORTRAN95 with ten compiler directives for scientific stream programming to facilitate inheriting legacy FORTRAN codes. These directives implicitly define basic streams, derived streams, stream-level programs and kernel-level programs, which concisely abstract the features of stream programming model.

There are 3 kinds of compiler directives: stream-level directives, kernel-level directives and optimization directives.

The stream-level directives include SBB and SBE, which are in the form of *!$SF95 streamblock begin* and *!$SF95 streamblock end* respectively. SBB and SBE implicitly define the scope of a stream-level program. The array variables between them are implicitly defined as basic streams, which will be stored in FT64's off-chip memory. Besides, SBB indicates the time when the input basic streams are loaded from the host memory while SBE indicates the time when the output basic streams are stored back to the host memory. The codes outside the scope of SBB and SBE (such as statement 1 and 2 in Figure 3) will be executed on the host. The stream-level analysis and optimization will be performed for the codes within SBB and SBE (such as statement 3 to 24 in Figure 3).

The kernel-level directives include KBB, KBE, KLP and SCP, which are in the form of *!$SF95 kernelblock begin, !$SF95 kernelblock end, !$SF95 kernelloop* and *!$SF95 streamcopy* respectively. Kernel-level programs are implicitly defined from the following cases: the statements between KBB and KBE, the loop statement after KLP and the vector statement after SCP. The array variable references in a kernel-level program are implicitly defined as derived streams, which will be loaded to the SRF from the off-chip memory. Statements 4~9, 10~12, 13~14 and 17~22 in Figure 3 are 4 kernel-level programs.

The optimization directives include SUR, KPL, KUR and RDT, which are in the form of *!$SF95 stream unroll n*, *!$SF95 kernel pipeline n, !$SF95 kernel unroll n* and *!$SF95 reduction op1(a, b) op2(c)*. Here, op1 and op2 are reductive types which can be + or *, while a, b and c are reductive variables of the corresponding types (a and b are of op1 type, and c is of op2 type.). SUR is used to optimize the execution of stream-level programs, and the other three are used to optimize the execution of kernel-level programs. SUR indicates the unrolling number of the loop in stream-level programs, such as statement 16 in Figure 3. KPL indicates the software pipeline's depth of the inner-most loop in kernel-level programs, such as statement 20 in Figure 3. KUR indicates the unrolling number of the inner-most loop in kernel-level programs, such as statement 19 in Figure 3. RDT indicates the reductive variables and their reductive types in kernel-level programs, such as statement 6 in Figure 3.

Note that KPL, KUR and RDT can only exist within kernel-level programs, whereas SUR can only exist within stream-level programs but outside kernel-level programs. SBB and SBE must appear in pairs, and so it is with KBB and KBE.

Figure 3 shows a complete SF95 program and its execution on FT64. The stream-level pseudo codes converted from statements 4~9 are given on the top right of Figure 3, and the bottom right of Figure 3 shows how the implicitly defined kernel-level program is executed on FT64, including how the derived streams are stored in SRF and how each cluster executes different iterations in the kernel-level program.

SF95 is used for programming one module. For programming multiple modules, MPI and SF95 mixed programming model is used. Figure 4 gives such a template. First, module 0 dispatches initial data to the other modules. Then, these modules concurrently execute different iterations of the loop. Finally, module 0 collects the results from the other modules.

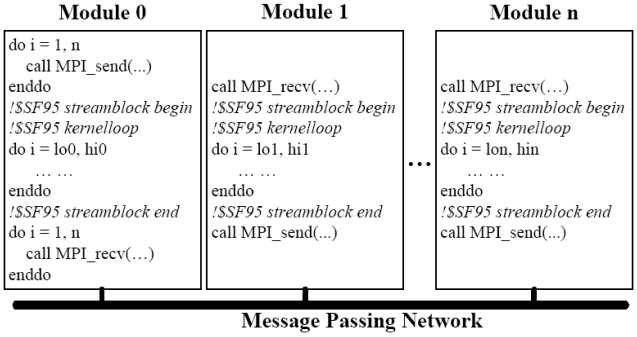

Figure 4. MPI and SF95 mixed programming template.

4. FT64 IMPLEMENTATION

FT64 is implemented with 49.2-million transistors on a 12*12mm^2 die in a 0.13-um process with a standard-cell design flow. Figure 5(a) shows the layout of the die. Clusters consume 35.8 percent of the die, SRF consumes 16.5 percent, and the others share the rest of the area. The chip is packaged into HSBGA with 1156 pins. FT64 works at 500MHz and the peak performance reaches 16GFLOPS. And its power consumption is 8.6w. Figure 5(b) is the top and bottom photographs of FT64 ("YH" is the logo of the design team and this is the version 2 of FT64). A basic module has a host and 8

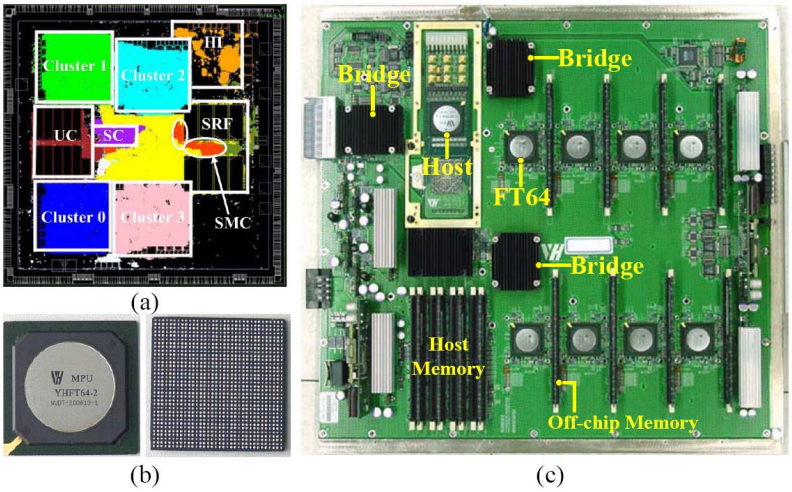

Figure 5. (a) Layout of FT64. (b) Top and bottom photographs of FT64. (c) Basic module.

Table 1. Stream-level instruction set

Name	Main Functions
MOVE	Move the data in the source register to the destination register
WRITE_IMM	Write a immediate value to the destination register
BARRIER	Insure the execution order of the instructions
RESET	Reset FT64
MEMOP	Transfer a stream between DRAM and SRF, addressing modes including sequential, strided, indexed, and bit-reversed
LOAD_UCODE	Load the micro code from SRF to UC storage
CLUSTOP	Start the execution of the kernel on the clusters
CLUSTER_RESTART	Restart the kernel with the input or output stream
SYNCH_UC	Synchronize the execution of the clusters
NETOP	Transfer a stream between the local SRF and remote SRF
NET_RESTART	Restart a stream to continue the network transfer

FT64s as shown in Figure 5(c). Multiple modules may compose a high performance computer system for scientific computing as shown in Figure 2.

We perform 64-bit extension and scientific computing oriented optimization on FT64 from the aspects of instruction set architecture, stream controller, micro controller, ALU cluster, memory hierarchy and interconnection interface.

4.1 Instruction Set Architecture

There are 11 stream-level instructions and 140 kernel-level instructions in FT64's instruction system.

Table 1 gives the stream-level instructions, which are used to control the transfer of streams and scalars as well as the execution on clusters.

The design of kernel-level instructions focuses on the supports for scientific computing, especially for double-precision floating-point calculations. There are 7 types of kernel-level instructions as shown in Table 2. The process of subnormal floating points by floating-point supporting instructions is compatible with the IEEE754 standard. The communication latency across clusters costs three cycles. The initial latency of reciprocal and square root operations costs two cycles, followed by iterations to obtain the required precision.

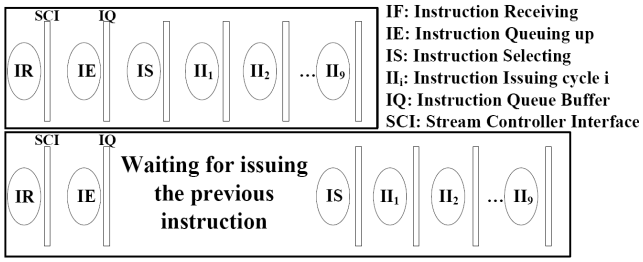

Figure 6. SC pipeline.

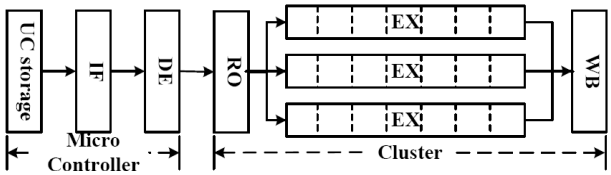

Figure 7. Kernel-level pipeline.

Table 2. Kernel-level instruction set

Instruction Type	#Instructions	Main Functions
FP/Integer MA ops	32	FP/Integer MA, integer/FP conversions.
FP miscellaneous ops	17	FP logic ops, FMAX, FMIN, FABS, FP test (=, !=, >, <, etc.) .
FP supporting ops	13	FP type check, FP's exponent and mantissa fetching & setting, FP normalizing, etc.
Divide/Square root ops	7	FP divide and square root ops.
Integer arithmetic & logical ops	28	Integer +/-, ABS, integer test (=, !=, >, <, etc.), integer logical ops (OR, AND, XOR and NOT), shift ops, conditional code/integer conversions.
Program control ops	9	Loop ops, conditional test, synchronization ops, etc.
Conditional and data communication ops	34	State initialization, state update, (conditional) stream input/output ops, buffer read and data communication, etc.

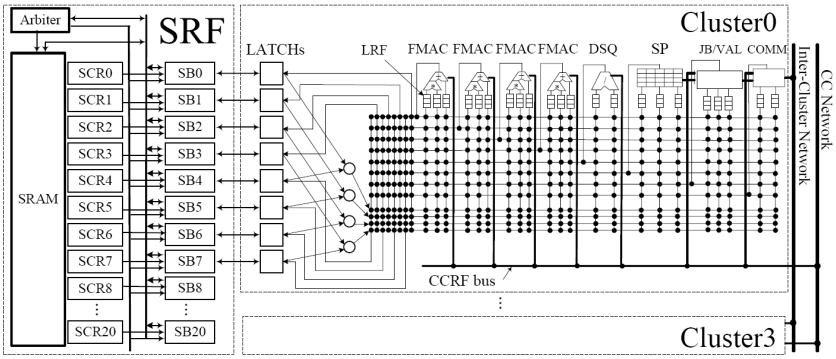

Figure 8. SRF and cluster's structure diagram.

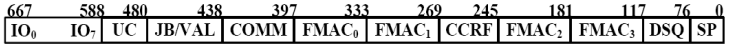

Figure 9. Kernel-level VLIW instruction.

4.2 Stream Controller and Micro Controller

The SC executes the stream-level instructions which are loaded to SC's instruction queue by the host through the HI. It is composed of a SC interface (SCI), a scoreboard, an instruction issuing unit and a SC register file (SCRF). The SC pipeline is shown in Figure 6. The instruction issuing unit can issue one stream instruction once, which costs at most 9 cycles. The SCRF is mapped to the addressing space of the host. The values can be transferred between the SCRF and other registers in FT64.

The UC controls the execution of kernel-level programs on the 4 clusters in a SIMD fashion. It consists of a UC storage (to store kernel-level instructions and scalars), a control unit, a micro-code register file, a condition register file, a software pipeline check logic, a conditional input/output flag queue and an input/output control logic. The UC storage is a 167KB dual-ported SRAM which can store 2K VLIW instructions. When the kernel-level program is executed, the UC issues kernel-level instructions to the clusters, and transfers scalars to/from them via a data bus.

The kernel-level pipeline of FT64 is shown in Figure 7. The stages of IF (instruction fetch) and DE (decode) reside in the UC, while the stages of RO (read operators), EX (execute) and WB (write back) reside in the clusters.

4.3 ALU Cluster

The four ALU clusters are shown in Figure 8. Data are transferred through inter-cluster network, and conditional codes are exchanged through CC network. Each cluster is composed of eight function units (FUs), eight local register files (LRFs), seven condition code register files (CCRFs) and an intra-cluster network. The FUs consist of four FMACs, a DSQ (divide/square root unit), a JB/VAL, a COMM (communication unit) and a 256-word scratch-pad memory (SP). The LRFs are used to buffer input and intermediate data. The CCRFs have the same values and are distributed in the FUs except the SP. Each CCRF has 16 8-bit-wide entries and contains one read and three write ports. The CCRFs, JB/VAL, COMM and SP cooperate to perform conditional routing and data communication among the clusters.

The kernel-level instructions are in the form of VLIW as shown in Figure 9. The first nine fields from bit 0 (the SP field) correspond to the eight FUs and CCRF, and the following two fields (UC, IO) are interpreted by the UC to control the execution of kernel-level instructions and the input/output of the intermediate results.

4.4 Memory Hierarchy

The memory hierarchy of FT64 has 3 levels: LRF, SRF and off-chip memory. The LRFs are distributed in the clusters' FUs, where the FMAC's LRF has 96 64-bit-wide entries, and the DSQ's has 64 64-bit-wide entries. The total capacity of the LRFs is 19KB and the total bandwidth is 544GB/s. All the LRFs can be bypassed. The register 0 and register 1 in the FMACs and DSQ are set to constants.

The SRF is used to buffer derived streams and kernel-level

programs. It is in the size of 256KB and divided into 2048 16-word blocks. As shown in Figure 8, the SRF is composed of SRAM banks, 21 stream buffers (SBs), an arbiter and 21 stream control registers (SCRs). The SRF can not be indexed. Virtual multi-ported access is implemented in the SRF via the SBs and centralized arbiter. Each SB contains 256 bytes of storage to allow for double buffering, which can effectively hide access latency. SB0~SB7 are connected to the 4 clusters separately with a total bandwidth of 64GB/s. SB8 is connected to the UC, SB9~SB12 are connected to the SMC and SB13~SB20 are connected to the NI.

The off-chip memory is controlled by the SMC and DDRMC, which cooperate to load and store derived streams. The SMC receives the access instructions from the SC, generates memory addresses, and sends read/write requests to the DDRMC. The DDRMC controls the access to the off-chip memory. The SMC has two address generators, two reordering stream buffers and sixteen memory address registers. It supports two simultaneous memory accesses. Four types of accesses are supported by the SMC: sequential, strided, indexed, and bit-reversed. For sequential access, the SMC generates burst access requests. The SMC operates at 500MHz with the bandwidth of 16GB/s to the SRF, and 8GB/s to the DDRMC. The DDRMC works at 200MHz with the bandwidth of 6.4GB/s.

4.5 Network Interface and Host Interface

The NI provides stream communication mechanisms that allow two FT64s to transfer streams. It provides 4 external bidirectional channels, each of which supports two virtual channels. The total bandwidth of the NI is 1.2GB/s. Figure 10 gives the packet format in stream communications. A stream is partitioned into multiple packets. In a packet, the *type* field specifies its location in the packet sequence (head, tail or middle), the *vc* field specifies the virtual channel number, the *len* field specifies the length of the data field, the *ctr* field specifies the control information needed in transfer, the *ri* field specifies the routing information, the *ri_backup* field is *ri*'s backup, the *ui* field specifies user-customized attributes, and the *data* field contains 32 32-bit-wide items.

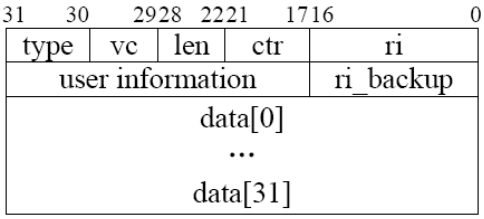

Figure 10. Packet format in stream communication.

To transfer streams, the SCs of the sender and receiver execute the *send*-type and *receive*-type *netop* stream instructions respectively. In detail, the sender's SC constructs the packets, sets up the connection between the specified SB and NI, and sends the packets. As soon as the receiver's NI receives the packets, it notifies the SC to set up the connection between the NI and SB, and the SC stores the packet data into the SRF.

The HI provides message passing communication mechanisms that allow block data transfers between the host and FT64s within a module, or between different modules. The total bandwidth of the HI is 2.4GB/s. The DBT protocol supports up to 64MB data per transfer. Figure 11 gives the packet format in message passing communication. The *CMD* field specifies the control commands, including the RRA and DBT commands, which will be processed by the RRA and DBT controllers in the HI respectively. The *Source* field specifies the source node, the *Dest* field specifies the destination node, the *Address* field specifies the storing address in the destination node, and the *Suppl* field specifies other information, such as the number of data items in the packet. The data field is composed of multiple 128-bit data items. An RRA packet has at most one data item and a DBT packet has at most 8 items.

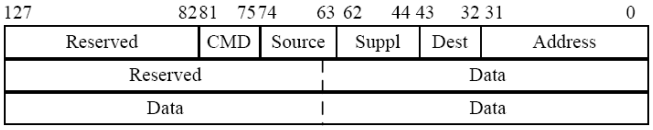

Figure 11. Packet format in message-passing communication.

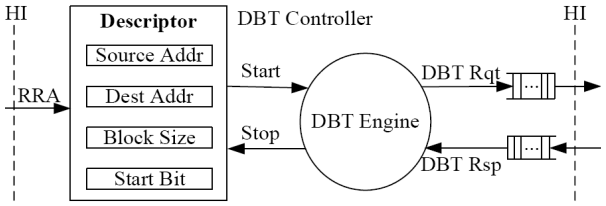

Figure 12. DBT controller.

Figure 12 gives the structure of the DBT controller, which contains a group of registers: the source address register, the destination address register, the block size register and the start register. In block data transfer, the host of the sender writes the above registers one by one via the RRA packets to start the DBT Engine. The DBT Engine constructs the control information in DBT packets, packs data into multiple continuous DBT packets, and sends them to the destination. When the receiver receives a DBT packet, a response message must be returned to the sender. When all data transfers are completed, the receiver's DBT Engine will set a complete flag for the host to query.

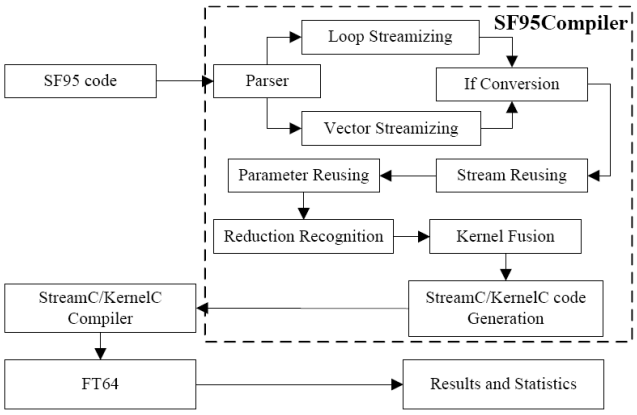

Figure 13. The workflow of the software system.

5. SF95COMPILER IMPLEMENTATION

FT64's programming environment mainly has 3 parts: a high-level FT64 programming language, SF95; a source-to-source compiler, SF95Compiler, to transform programs from SF95 to StreamC/KernelC; and a low-level runtime library based StreamC/KernelC programming interface and its compiler. In this section, we will focus on the implementation of SF95Compiler. Its workflow is depicted in the black dashed rectangle area of Figure 13. SF95Compiler is designed based on G95's front-end, including 9 steps: syntax parsing, loop streamizing, vector streamizing, *if*

conversion, stream reusing, parameter reusing, reduction recognition, kernel fusion and StreamC/KernelC code generation. SF95Compiler also uses traditional compiler transformations [27], such as loop unrolling and loop fusion.

5.1 Loop Streamizing

Loop streamizing automatically transforms the loop statements following KLPs into intermediate representation for kernel-level programs. In detail, SF95Compiler first generates basic streams from the array variables of each assign statement within a loop, and inserts *streamLoad* (load data from arrays to basic streams) statements before the loop and *streamStore* (store data from basic streams back to arrays) statements behind the loop. Then, SF95Compiler generates derived streams from the array references within the loop, which functions as the parameters of the corresponding kernel-level program. Next, it generates *loop_stream*/*loop_count* statements in the kernel-level program, and replaces the original loop with the kernel-level program. Scalar variables in the loop are passed to the kernel-level program as UC variables and stored in the UC storage.

Take statements 4~9 in Figure 3 as an example, the corresponding loop following statement 4 is transformed to the kernel-level program called *kf_0* with the StreamC/KernelC pseudo codes on the top right corner of Figure 3, where *s_a* and *s_b* are the corresponding basic streams of arrays a and *b*, the derived stream *s_a(0, 128)* is the input stream of the kernel-level program, *s_b(0, 128)* is the output stream, and *uc_sum* is the UC variable corresponding to *sum*.

5.2 Vector Streamizing

Vector streamizing automatically transforms the vector statements between KBB and KBE into intermediate representation for kernel-level programs. As vector statements can always be equivalent to a certain or several loop(s), vector streamizing is similar to loop streamizing, except that variable renaming might be performed to preserve the original dependences. Statements 10~12 in Figure 3 can be transformed to the kernel-level program call *kf_1(s_b(0, 127, stride, 2)*, *s_b(1, 128, stride, 2)*, *s_c(0, 64))*.

5.3 Stream Reusing

The main idea of stream reusing is to generate as few basic streams as possible. If different statements of a stream-level program access the same array variable, it is unnecessary to keep multiple copies for this array (or basic stream) in FT64's off-chip memory. This avoids memory consistency maintenance between different basic streams and reduces the capacity requirement of FT64's off-chip memory. Therefore, stream reusing detects all the references to the same array variable and generates only one basic stream with minimum data amount and covering all referenced sections. In this way, different references to this array variable are transformed into different derived streams of the corresponding basic stream. For example, there are 3 references to array *b* in statement 7 and 11 in Figure 3. After stream reusing, only one basic stream *s_b* is generated and the 3 corresponding references to *s_b* are *s_b(0, 128)*, *s_b(0, 127, stride, 2)* and *s_b(1, 128, stride, 2)*.

5.4 Parameter Reusing

The main idea of parameter reusing is to keep as few derived streams in SRF as possible, i.e. generate as few input/output stream parameters for kernel-level program as possible. For architecture constrains, there are limited number of derived streams allowed in the SRF (at most 8, corresponding to SCR0~7). By means of parameter reusing, SCR's utility can be reduced as a preparation for kernel fusion. Meanwhile, if derived streams overlap, parameter reusing can improve SRF's data locality.

Parameter reusing can be subdivided into input reusing, output reusing and output-input reusing. The two input streams *s_b(0, 127, stride, 2)* and *s_b(1, 128, stride, 2)* of statement 11 in Figure 3 can be reused into one input stream *s_b(0, 128)*. What's more, if kernel fusion(described in Section 5.5) is performed, the reused input stream *s_b(0, 128)* of statement 11 in Figure 3 can be output-input reused with the derived stream *s_b(0, 128)* corresponding to the array reference *b(i)* of statement 7 in Figure 3. In fact, if two output streams are in the same form within a kernel, by means of output reusing, only the last output stream parameter is kept. Note that as a stream in the SRF cannot be both input and output in one kernel-level program, for two stream parameters of which the first one appears as input and the second one as output, even if they are in the same form, they cannot be reused.

5.5 *If* Conversion

If conversion, which transforms *if* statement to FT64's *select* statement, is mainly based on the theories brought forward by Randy Allen et al. to transform control dependence into data dependence [2]. In detail, SF95Compiler first arranges the statements in both its *true* and *false* branches, and then generates the corresponding conditions for each branch. Finally, SF95Compiler constructs a set of *select* statements according to the conditions and return values in both branches.

5.6 Reduction Recognition

Reduction operations can be accelerated by FT64's inter-cluster communications. Therefore, for the reductive variables and types defined by RDT in kernel-level programs (e.g. statement 6 in Figure 3), SF95Compiler can map them directly to the corresponding form in StreamC/KernelC to accelerate its execution.

5.7 Kernel Fusion

Kernel fusion can improve compute intensity under the limitation of at most 8 stream parameters. Before kernel fusion, statement reordering is performed to make kernel-level programs as continuous as possible. For two adjacent kernel-level programs K_1 and K_2, if they satisfy the condition of loop fusion and the total number of stream parameters is not more than 8, they can be fused to one kernel-level program. For example, the first two kernel-level programs corresponding to statements 4~9 and statements 10~12 in Figure 3, can be fused to one kernel-level program.

6. EXPERIMENTAL RESULTS

To verify whether FT64 is suitable for scientific computing and validate the effectiveness of our programming environment, we perform tests on 9 typical scientific application kernels as specified in Table 3. NLAG-5 is a nonlinear algebra solver of two-dimensional nonlinear diffusion of hydrodynamics.

The following experiments are tested on both the FORTRAN versions and SF95 versions of the above 9 programs. The original FORTRAN programs are compiled by Intel's compiler *ifort* with the optimization option *-O3*, and then executed on a single-core Itanium 2 server. Itanium 2 runs at 1.6GHz and the sizes of the caches are 16KB for the L1 cache, 256KB for the L2 cache and

Table 3. Specifications of 9 benchmarks.

Name	Swim	EP	MG	CG	FFT	Laplace	Jacobi	GEMM	NLAG-5
Source	Spec2000	NPB	NPB	NPB	-	NCSA	-	BLAS	-
#Arrays	14	1	3	2	1	1	4	2	2
Prob. Size (doubles)	513×513	131072	64×64×64	500×500	4096	256×256	128×128	256×256	256×256

Table 4. Performance speedups of FT64 to Itanium 2.

Tests	Swim	EP	MG	CG	FFT	Laplace	Jacobi	GEMM	NLAG-5
Performance Speedup	1.04	2.55	1.35	0.10	8.01	2.41	1.04	1.97	0.97

6MB for the L3 cache. There is also a 4GB off-chip memory with the bandwidth of 6.4GB/s. The SF95 programs are executed on not only a module of the FT64 system but also a cycle-accurate simulator of FT64 module. By using the simulator, we can obtain the programs' memory access characteristics as shown in Figure 14(b), 15(a) and 15(b).

The execution time is obtained by inserting the clock-fetch assembly instructions. If the data size of the program is small, we eliminate the extra overheads (such as system calls) by means of executing it multiple times and calculating the average time consumption. As I/O overheads are hidden in our experiments, the CPU time is nearly equal to the wall-clock time.

Table 4 shows the performance speedups of the 9 programs on FT64 vs. on Itanium 2. It can be observed that there are 4 programs (EP, FFT, Laplace and GEMM) that perform better on FT64 than Itanium 2, 4 programs (Swim, MG, Jacobi and NLAG-5) that perform comparably, and 1 program (CG) that performs relatively worse. FFT enjoys the highest speedup. It is because that FFT's data access pattern in the butterfly and bit-reverse transformations has special supports from FT64, but none from Itanium 2. For EP's second highest speedup, it is because that the intensive computation in EP can well exploit FT64's powerful FMACs. For CG's lowest speedup, it is because that its effective execution time (computation time plus memory access time) only takes up a small part of the total execution time. Most of the time is consumed in SRF allocation and memory access preparation, for the short length of the derived streams and irregular memory access pattern in CG.

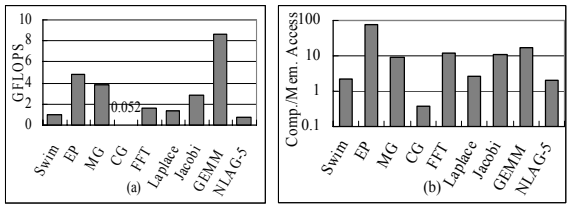

Figure 14. (a) Performance of tests on FT64.
(b) Ratios of computation to memory access on FT64.

Figure 14(a) presents the performance of the tests on FT64 measured in GFLOPS. The results show that, the sustained performance of compute-intensive programs (EP, MG, GEMM, Jacobi and FFT) is 10.3%~54.3% of FT64's peak performance; the sustained performance of memory-intensive programs (Swim, Laplace and NLAG-5) is 4.4%~8.5% of FT64's peak performance; while the sustained performance of sparse-matrix programs (CG) is only 0.33% of FT64's peak performance, which is poorer on FT64 than on Itanium 2.

Figure 14(b) shows the ratios of computation to memory access on FT64. For such compute-intensive programs as EP, MG, GEMM, Jacobi and FFT, we should first optimize the instruction-level parallelism and data-level parallelism in their kernel-level programs. For such memory-intensive programs as Swim, CG, Laplace and NLAG-5, we should first consider to reduce the overhead in memory accesses.

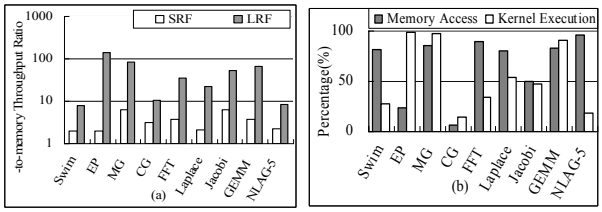

Figure 15. (a) SRF- and LRF-to-memory throughput ratios.
(b) Percentages of memory access and kernel execution.

SRF (or LRF) -to-memory throughput ratio is the ratio of the data throughput in the SRF (or LRF) to that in the off-chip memory. Figure 15(a) shows the SRF/LRF-to-memory throughput ratios when programs are running on FT64. It can be observed that EP, MG, GEMM, Jacobi and FFT achieve higher LRF-to-memory throughput ratios, which are 141:1, 78:1, 67:1, 57:1 and 35:1 respectively. These compute-intensive programs can well exploit LRF locality. But Swim and NLAG-5's LRF-to-memory throughput ratios are not high. That is because these programs are limited by memory access and their LRF locality is poor. MG, GEMM, Jacobi and FFT achieve higher SRF-to-memory throughput ratios, which means they well exploit SRF locality. But Swim, EP, Laplace and NLAG-5's SRF-to-memory throughput ratios are not high, which means these programs poorly exploit SRF locality.

The overlap between computation and memory access is another important factor that impacts a stream processor's performance. Figure 15(b) demonstrates the distribution of memory access time and kernel execution time when programs running on FT64, i.e. what percent the two parts take up as to the total program execution time. For compute-intensive programs such as EP, GEMM and MG, if memory access can be well hidden by computation, kernel's execution time will approximate the total execution time; and for memory-intensive programs such as Swim, NLAG-5 and Laplace, if computation can be well hidden by memory access, memory access time will approximate the total execution time. CG is a special case, as the lengths of its derived streams are so short that most of the time is taken up by SRF allocation and memory access preparation. As a result neither its memory access nor computation takes up much time in the total execution.

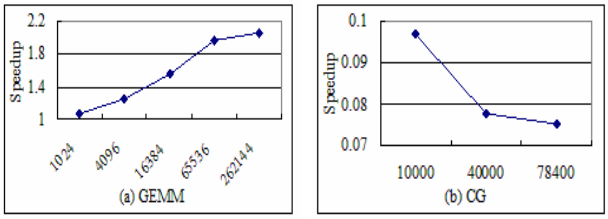

Figure 16. Effects of varying the data sizes of GEMM (a) and CG (b) on performance speedups.

To evaluate the scalabilities of scientific programs on FT64, we vary the data sizes of GEMM and CG to observe the effects on their performance speedups. Figure 16(a) shows that with the increase of data size, GEMM's speedup on FT64 is increased, but the speed of increase becomes slower and slower. In fact, when the data size is 32*32, the serial part of the stream program has taken up more than 40% of the total execution time; when the data size reaches 256*256, the serial part takes less than 0.6% of the total execution time, and FT64's speedup is approximately 2 as compared with Itanium 2. When the data size keeps increasing, the execution time of the serial part can be almost ignored. The performance speedup trend of the other programs except CG is similar with that of GEMM. Figure 16(b) shows that with the increase of data size, CG's performance speedup is decreased. It can be concluded that the scalabilities of all the programs except CG are good on FT64.

The above results show that FT64, as a representative stream processor, is suitable for developing most scientific applications with the supports of SF95 and SF95Compiler.

7. CONCLUSION AND FUTURE WORK

In this paper, we study whether stream processor is suitable for scientific computing from three aspects: architecture, programming language and compiler. We design and implement a 64-bit stream processor, FT64, for scientific computing. We carry out 64-bit extension design and scientific computing oriented optimization on instruction set, stream controller, micro controller, clusters, memory hierarchy and interconnection interface. We bring forward two kinds of communications and an interconnection to facilitate the design of FT64-based high performance computers. We design a FORTRAN 95 based stream language, SF95, and its compiler, SF95Compiler, with the adoption of some novel compiler optimizing techniques including loop streamizing, vector streamizing, stream reusing, parameter reusing, *if* conversion, reduction recognition and kernel fusion, etc. We perform experiments with nine typical scientific application kernels on FT64, and the results show that FT64 performs equal or better to Itanium 2 for the programs except CG.

In the future, our efforts will mainly focus on the following aspects: Carry out researches for the technologies to build FT64-based peta-flops high performance computer; improve FT64's micro-architecture (such as, supporting the indexed access in the SRF, and providing more types of stream access) and the performance of FT64 chip (such as, increasing the number of FMAC units from 16 to 32 or 64, and increasing the frequency to 1GHz); keep on studying multi-stream inter-chip programming and compiler optimization techniques; test more scientific benchmarks and applications.

8. ACKNOWLEDGMENTS

We would like to thank Jing Du, Li Wang, Canqun Yang, Yuhua Tang, Chunjiang Li, Guibin Wang, Tao Tang, Kun Zeng, Xiangli Qu for their efforts on this work. We also thank Jun Gao, Yong Li, and all other members of this research group.

This work was supported by NSFC (60621003).

9. REFERENCES

[1] Agrawal, S., Thies, W. and Amarasinghe, S. Optimizing Stream Programs Using Linear State Space Analysis. In *CASES '05*, 2005, 126-136.

[2] Allen, J. R., Kennedy, K., Porterfield, C. and Warren, J. Conversion of Control Dependence to Data Dependence. In *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, 1983, 177-189.

[3] Bove, V. M. and Watlington, J. A. Cheops: A Reconfigurable Data-Flow System for Video Processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5, 2, (1995), 140-149.

[4] Buck, I. Brook Spec v0.2. Report of Stanford University, http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf, 2003.

[5] Burger, D., Keckler, S. W., McKinley, K. S., Dahlin, M., John, L. K., Lin, C., Moore, C. R., Burrill, J., McDonald, R. G. and Yoder, W. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37, 7(2004), 44-55.

[6] Caspi, E., DeHon, A. and Wawrzynek, J., A Streaming Multi-Threaded Model. In *Proceedings of the Third Workshop on Media and Stream Processors* (2001), 21-28.

[7] Dally, W. J., Hanrahan, P., Erez, M. and Knight, T. J. Merrimac: Supercomputing with Streams. In *SC2003,* Nov 2003.

[8] Dou, Y. and Lu, X. C. LEAP: A Data Driven Loop Engine on Array Processor. In APPT'03: Proceedings of the 5th International Workshop on Advanced Parallel Processing Technologies, 2003, 12-22.

[9] Gordon, M. I., Thies, W., Karczmarek, M., Lin, J., et al., A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS-X* (2002), 291-303.

[10] Hoare, T. Communicating Sequential Processes. *Communications of the ACM*, 8, 21(1978), 666-677.

[11] Kapasi, U., Dally, W. J., Rixner, S., Owens, J. D. and Khailany, B. The Imagine Stream Processor. In *ICCD'02: Proceedings of 20th IEEE International Conference on Computer Design*, 2002, 282-288.

[12] Kapasi, U. J., Mattson, P., Dally, W. J., Owens, J. D. and Towles, B. Stream Scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, 2001, 101-106.

[13] Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P. and Owens, J. D. Programmable Stream Processors. *IEEE Computer*, 36, 8 (Feb, 2003), 54-62.

[14] Kozyrakis, C. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, University of California at Berkeley, 2002.

[15] Mattson, P. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[16] Mattson, P., Dally, W. J., Rixner, S., Kapasi, U. J. and Owens, J. D. Communication Scheduling. *SIGPLAN Not*, 35, 11(2000), 82-92.

[17] May, D. OCCAM. *SIGPLAN Notices*, 18, 4(1983), 69-79.

[18] Owens, J., Kapasi, U., Mattson, P., Towles, B., Serebrin, B., Rixner, S. and Dally, W. Media Processing Applications on the Imagine Stream Processor. In *ICCD'02*(2002), 295-302.

[19] Pham, D., Asano, S., Bolliger, M., Day, M. N., Hofstee, H. P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., J.Warnock, Weitzel, S., Wendel, D., Yamazaki, T. and K.Yazawa, a. The Design and Implementation of a First-Generation Cell Processor. In *ISSCC'05*, 2005, 184-185.

[20] Rixner, S. Stream Processor Architecture. Kluwer Academic Publishers Group, ISBN: 0-7923-7545-9, 2002.

[21] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P. and Owens, J. D. Memory Access Scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000, 128-138.

[22] Taylor, M., Kim, J., Miller, J., Wentzlaff, D., et al., The RAW Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 22,2 (2002), 25-35.

[23] Thies, W., Karczmarek, M. and Amarasinghe, S. StreaMIT: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, 2002, 179-196.

[24] Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., H., H. o., Brown, M. and Amarasinghe, S. StreamIt: A Compiler for Streaming Applications. MIT-LCS Technical Memo TM-622, Cambridge, MA, http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TM-622.pdf, 2001.

[25] Wen, M., Wu, N., Xun, C., Wu, W. and Zhang, C. Analysis and Performance Results of a Fluid Dynamics Application on MASA Stream Processor. In *ICIS'06: Proceedings of International Conference on Information Systems*, 2006, 350-354.

[26] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P. and Yelick, K. The Potential of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, 2006, 9-20.

[27] Yang, X., Du, J., Yan, X. and Deng, Y. Matrix-Based Programming Optimization for Improving Memory Hierarchy Performance on Imagine. In *ISPA'06*, 2006.